WindRiver SYSTEMS
An ISO 9001 Registered Company

WTN-49

The Peripheral Component Interconnect (PCI) Bus and vxWorks

A Discussion of the implementation of PCI support on Tornado/vxWorks BSPs.

Wind Tech Notes are maintained and published by Wind River Systems Customer Support

	Telephone	Email	Fax
Corporate	800/872-4977 Toll Free, US and Canada.	support@wrs.com	510/749-2164
Europe	(+) 800 49 77 49 77	support@wrsec.fr	(+) 33 1 60 92 63 15
Japan	011-81-3-5467-5900	support@kk.wrs.com	011-81-5467-5877

If you purchased your Wind River systems Product from a distributor, please contact your distributor to determine how to reach your technical support organization.

Please provide your license number when contacting Customer Support

Overview

WRS provides support for the Peripheral Component Interconnect (PCI) bus in many of its newer Board Support Packages (BSPs). It is the intent of this paper to provide a general outline of WRS implementation of PCI under vxWorks.

PCI Address Spaces and Memory Mapping

The PCI Bus provides three types of address space: I/O, Memory and Configuration. Each device is mapped to memory and/or I/O space through Base Address Registers located in configuration space. This eliminates the need for hardware jumpers to determine the addresses for the board. The configuration of the PCI bus is almost completely controlled by software registers in configuration space.

Therefore each PCI device must be configured before it can be used. This means that its memory or I/O address must be assigned and the device must be enabled to respond to normal PCI transactions.

WRS provides a standard library for accessing configuration space called pciConfigLib.c. This module supports Host-Bridge devices using access mechanism 1 and access mechanism 2 as defined in the PCI Spec 2.1. A third access mechanism that WRS refers to, as mechanism 0 is available for non-standard bridges. This mechanism 0 calls out to special BSP specific routines to perform configuration read and write functions, while maintaining the same user interface of the other mechanisms.

The pciConfigLib.c module provides routines for accessing any register within configuration space. It also includes routines to scan a bus looking for instances of a particular device or a particular class. There is also a routine that can configure simple devices with a simple calling interface.

PCI Interrupt Handling

The PCI specification does not address how interrupt signals are routed to the interrupt controller device for a motherboard bus. Each device has 4 interrupt pins available. They are named A, B, C, and D. Each single interrupt PCI device is required to always use Int Pin A to generate an interrupt. Devices with multiple functions can assign 1 interrupt pin per function. If a device implements all 8 possible sub-functions, there would be 2 interrupt sources on each interrupt pin. A PCI interrupt handling system needs to be able to call several interrupt service routines for each generated interrupt. The normal operation is to call all interrupt handlers each time. Each handler is responsible for checking that its associated device is actually generating an interrupt. If it isn't the handler returns immediately so that the next interrupt handler can be called.

The module pciIntLib.c provides for multiple interrupt handlers to be attached to a single interrupt line. This is done by installing a special handler that calls all of the routines from a linked list. The pciIntConnect() and pciIntDisconnect() functions simply add or delete handlers from the linked list.

PCI Configuration Strategies for vxWorks.

The macro INCLUDE_PCI should be defined to indicate that the BSP includes PCI support.

The macro PCI_CFG_TYPE should be assigned to one of the following values:

1. Static configuration: PCI_CFG_FORCE

   The programmer/bsp writer programs each device separately through data tables, configuration macros, or some other method. The addresses and interrupt numbers for each device are known in advance and are programmed directly to the device.

2. Dynamic Configuration: PCI_CFG_AUTO

This is the normal PCI dynamic configuration where the PCI bus is scanned and memory or I/O addresses are assigned dynamically to each device found. This is typical X86 BIOS PCI initialization routines. The programmer do not need to know what addresses get assigned to a device beforehand. It is sufficient that the correct amount of memory or I/O space is assigned.

This functionality is implemented in the WRS pciAutoConfigLib module.

3. No Configuration: PCI_CFG_NONE

This is reserved for those situations where an agent outside of vxWorks performs configuration. In this case, all the PCI devices are preconfigured before vxWorks is started. The difficulty with this method is the vxWorks system does not have full knowledge of the range of addresses and bus numbers that were assigned during the scan. This is usually necessary for dynamic set up of the MMU tables when the MMU is in use.

PCI Initialization Sequences

When vxWorks is started, the first use of any device occurs just after the call to sysHwInit2(). However, the MMU memory map is initialized and activated between the calls to sysHwInit() and sysHwInit2(). Because of this, the recommended initialization sequence is:

1. sysHwInit () - The default MMU table entries should correspond to the access windows of the Host bridge that map local transactions to PCI transactions ('master' windows).

2. sysHwInit2 () – In sysHwInit2() the programmer either statically configures all the devices intended for use, or invokes the dynamic configuration tool pciAutoConfig().

Device Driver Initialization

The old paradigm for vxWorks drivers has the device create function (xxxDevCreate()) being called with the device addresses and other information passed as arguments. In the PCI world, this can only work with static configuration (PCI_CFG_FORCE). In the other two configuration models, the addresses and information are not known at compile time. In these cases, the device configuration information must be accessed after configuration is complete and then passed to the driver. Any older driver can be used with PCI devices if there is code to read configuration information from the device and then pass that information to the driver.

If a driver were specific to PCI devices it would be much more logical to specify a device by its configuration address, rather than memory or I/O addresses. However, PCI configuration addresses are not absolute or permanent.

If the programmer chose bus, device, and function numbers as the address reference, the problem would lie in the bus numbering. Bus numbers are assigned dynamically as the bus is scanned. A device that might be (2,5,0) at one point could easily be (1,5,0) at some other time depending on the insertion/removal of a device using a PCI-PCI bridge.

If a programmer chose a reference model using the vendor/device Id and instance number there is a similar lack of absoluteness. For example: Consider a case where the first instance of a device has a vendor/device Id 0x10110008. Again, if a card is inserted/removed, a device that was the first instance of device id 0x00011234 could now be the second instance of the same device. There exists no way to refer to devices on any bus, except bus 0, with a consistent and absolute address that never changes. However, this latter model of reference is the preferred one, since all that can change is the instance number of the

same type of device. In the first case, if a device changes from (2,5,0) to (3,5,0) and the system isn't aware of the change, the driver would attempt to control whatever device was in the old slot id of (2,5,0).

The paradigm for a PCI device driver should be to access the device through configuration space and determine the addresses and information need to control the device. With this information the driver is now quite traditional in its control of the device. It should not need to refer to configuration space during normal device operation. Most new PCI devices map all their configuration registers into memory space as well as any device specific registers. It is usually more efficient to access I/O or Memory space than configuration space.

Dynamic MMU mapping

To follow the model set for VME window mapping, PCI master and slave accesses should follow a standard set of macros for describing the windows between buses. It takes 3 values to describe any window between two busses.

PCI Master Access Windows

The first piece of information is the window address on the host side. The second piece of information is the window address on the remote side. The third is the size of the window. By convention WRS will use macros of the form "PCI_xxxx_LOCAL" to be the host side information. The form "PCI_xxxx_BUS" will represent the remote bus address. Macros of the form "PCI_xxx_SIZE" will describe the size of the window in bytes. Setting the size macro to zero should disable the corresponding window.

There are 3 natural windows from a memory-mapped perspective into an average Host-Bridge. One window will map local memory access to a PCI I/O access. One window to map local memory access to PCI MEMIO access (non-prefetchable). The third maps local memory accesses to PCI MEM accesses (prefetchable). Since there is no fully defined specification for host bridges, there certainly can be more windows than this. Several of the Host-bridge devices that have been seen also have a special window for PCI-IACK signaling.

The following is a typical excerpt from config.h in a typical PCI capable BSP:

```
/* Master window allows CPU to access PCI I/O addresses */

#define PCI_MSTR_IO_LOCAL           0xC0000000
#define PCI_MSTR_IO_BUS       0x00000000
#define PCI_MSTR_IO_SIZE            0x00010000

/* Master window allows CPU to access PCI Memory addresses (prefetch) */

#define PCI_MSTR_MEM_LOCAL          0x80000000
#define PCI_MSTR_MEM_BUS            0x00000000
#define PCI_MSTR_MEM_SIZE           0x01000000

/* Master window allows CPU to access PCI Memory  (non-prefetch) */

#define PCI_MSTR_MEMIO_LOCAL        0x82000000
#define PCI_MSTR_MEMIO_BUS          0x00000000
#define PCI_MSTR_MEMIO_SIZE         0x01000000

/* Master window allows CPU to generate PCI_IACK cycles */

#define PCI_MSTR_IACK_LOCAL         0x8e000000
#define PCI_MSTR_IACK_BUS           0x0e000000
```

```
#define PCI_MSTR_IACK_SIZE          0x100
```

There are 3 address spaces in PCI: memory, I/O, and configuration. MEM and MEMIO are not different address spaces they are different bus operations. The MEMIO is a simple register read/write. The MEM accesses will be turned into cache type of operations where memory will be read ahead of time (pre-fetched) and data is written with the 'write and invalidate' operation. This tells the target device that more data is coming and it should just empty its cache line since it is going to be completely rewritten by the time this operation is done. The PCI specification does not specify how the Host Bridge maps MEM accesses to optimized PCI bus transactions. This is Host-Bridge specific and the appropriate manual for Host-Bridge device should be consulted.

In order to access a PCI memory location, the local CPU must know what local address is mapped to that PCI location. The CPU must use the local address, not the PCI address of the device location. The formula to translate a PCI address into the corresponding address on the local bus is: (These examples do not check that the PCI address actually lies within the window boundaries)

Local Addr = PCI addr + (PCI_MSTR_XXX_LOCAL - PCI_MSTR_XXX_BUS)

For example:

```
#define PCI_MEM2LOCAL(x) \
     ((int)(x) + PCI_MSTR_MEM_LOCAL - PCI_MSTR_MEM_BUS)

#define PCI_MEMIO2LOCAL(x) \
     ((int)(x) + PCI_MSTR_MEMIO_LOCAL - PCI_MSTR_MEMIO_BUS)

#define PCI_IO2LOCAL(x) \
     ((int)(x) + PCI_MSTR_IO_LOCAL - PCI_MSTR_IO_BUS)
```

PCI Slave Windows

Each of the windows described above is a master window. The host CPU is the bus master for all transactions using that window. A slave window is one where the host board acts as the target device to a transaction initiated by some other device on the bus. The number and types of slave windows varies, but almost all will have at least one slave window that makes the local memory accessible to other PCI devices such as DMA bus masters or other CPUs.

```
/*
 * Slave window that makes local memory visible to PCI
 * devices
 */

PCI_SLV_MEM_LOCAL       /* Local address of window */
PCI_SLV_MEM_BUS         /* PCI Bus address of window */
PCI_SLV_MEM_SIZE        /* window size,0 means disabled */

/*
 * For X86 it is possible to have a slave window mapping
 * PCI IO to Local IO
 */
```

```
PCI_SLV_IO_LOCAL        /* Local address of window */
PCI_SLV_IO_BUS          /* PCI Bus address of window */
PCI_SLV_IO_SIZE         /* window size,0 means disabled */
```

It is possible to have slave windows that would map PCI I/O transactions to local memory or, in the case of the x86, to local bus I/O requests.

In order to pass a memory address to a remote device, the local address must be translated to its equivalent PCI address for the remote device to use. The formula for translating a local address to its PCI equivalent is: (This example does not check that the address lies within the window or not)

PCI addr = Local Addr + (PCI_SLV_XXX_BUS - PCI_SLV_XXX_LOCAL)

For example:

```
#define LOCAL2PCI_MEM(x) \
     ((x) + PCI_SLV_MEM_BUS - PCI_SLV_MEM_LOCAL)
```